The ryg blog
When I grow up I'll be an inventor.

A trip through the Graphics Pipeline 2011, part 9

July 12, 2011
This post is part of the series ***"A trip through the Graphics Pipeline 2011" (https://fgiesen.wordpress.com/2011/07/09/a-trip-through-the-graphics-pipeline-2011-index/)***.

Welcome back! This post deals with the second half of pixel processing, the "join phase". The previous phase was all about taking a small number of input streams and turning them into lots of independent tasks for the shader units. Now we need to fold that large number of independent computations back into one (correctly ordered) stream of memory operations. As I already did in the posts on rasterization and early Z, I'll first give a quick description of what needs to be done on a general level, and then I'll go into how this is mapped to hardware.

Merging pixels again: blend and late Z

At the bottom of the pipeline (in what D3D calls the "Output Merger" stage), we have late Z/stencil processing and blending. These two operations are both relatively simple computationally, and they both update the render target(s) / depth buffer respectively. "Update" operation here means they're of the read-modify-write variety. Because all of this happens for every quad that makes it this far through the pipeline, it's also bandwidth-intensive. Finally, it's order-sensitive (both blending and Z processing need to happen in API order), so we need to make sure to sort processed quads into order first.

I've already explained Z-processing, and blending is one of these things that work pretty much as you'd expect; it's a fixed-function block that performs a multiply, a multiply-add and maybe some subtractions first, per render target. This block is kept deliberately simple; it's separate from the shader units so it needs its own ALU, and we'd really prefer for it to be as small as possible: we want to spend our chip area (and power budget) on ALUs in the shader units, where they benefit every code that runs on the GPU, not on a fixed-function unit that's only used at the end of the pixel pipeline. Also, we need it to have a short, predictable latency: this part of the pipeline needs to process data in-order to be correct. This limits our options as far as trading throughput for latency is concerned; we can still process quads that don't overlap in parallel, but if we e.g. draw lots of small triangles, we'll have multiple quads coming in for every screen location, and we'd better be able to write them out as quickly as they come, or else all our massively parallel pixel processing was for nought.

Meet the ROPs

ROPs are the hardware units that handle this part of the pipeline (as you can tell by the plural, there's more than one). The acronym, depending on who you asks, stands for "Render OutPut unit", "Raster Operations Pipeline", or "Raster Operations Processor". The actual name is fairly archaic – it derives from the days of pure 2D hardware acceleration, with hardware whose main purpose was to do fast **Bit blits (http://en.wikipedia.org/wiki/Bit_blit)**. The classic 2D ROP design has three inputs – the current (destination) pixel value in the frame buffer, the source data, and a mask input – then computes some function of the 3 values and writes the results back to the frame buffer. Note this is before true color displays: the image data was usually in bit plane format and the function was some binary logic function. Then at some point bit planes died out (in favor of "chunky" representations that keep the bits for a pixel together), true color became the norm, the on-off mask was replaced with an alpha channel and the bitwise operations with blends, but the name stuck. So even now in 2011, when about the last remnant of that original architecture is the "logic op" in OpenGL, we still call them ROPs.

So what do we need to do, in hardware, for blend/late Z? A simple plan:

1. Read original render target/depth buffer contents from memory – memory access, long latency. Might also involve depth buffer and render target decompression! (I'll explain render target compression later)
2. Sort incoming shaded quads into the right (API) order. This takes some buffering so we don't immediately stall when quads don't finish in the right order (think loops/branches, discard, and variable texture fetch latency). Note we only need to sort

based on primitive ID here – two quads from the same primitive can never overlap, and if they don't overlap they don't need to be sorted!

3. Perform the actual blend/late Z/stencil operation. This is math – maybe a few dozen cycles worth, even with deeply pipelined units.
4. Write the results back to memory again, compressing etc. along the way – long latency again, though this time we're not waiting for results so it's less of a problem at this end.

So, build the late-Z/blending unit, add some compression logic, wire it up to memory on one side and do some buffering of shaded quads on the other side and we're done, right?

Well, in theory anyway.

Except we need to cover the long latencies somehow. And all this happens for *every single pixel* (well, quad, actually). So we need to worry about memory bandwidth too… memory bandwidth? Wasn't there something about memory bandwidth? Watch closely now as I pull a bunny out of a hat after I put it there way back in **part 2 (https://fgiesen.wordpress.com/2011/07/02/a-trip-through-the-graphics-pipeline-2011-part-2/)** (uh oh, that was more than a week ago – hope that critter is still OK in there…).

Memory bandwidth redux: DRAM pages

In part 2, I described the 2D layout of DRAM, and how it's faster to stay within a single row because changing the active row takes time – so for ideal bandwidth you want to stay in the same row between accesses. Well, the thing is, single DRAM rows are kinda large. Individual DRAM chips go up into the Gigabit range in size these days, and while they're not necessarily square (in fact a 2:1 aspect ratio seems to be preferred), you can still do a rough calculation of how many rows and columns there would be; for 512 Megabit (=64MB), we'd expect something like 16384×32768, i.e. a single row is about 32k bits or 4k bytes (or maybe 2k, or 8k, but somewhere in that ballpark – you get the idea). That's a rather inconvenient size to be making memory transactions in.

Hence, a compromise: the page. A DRAM page is some more conveniently sized slice of a row (by now, usually 256 or 512 bits) that's commonly transferred in a single burst. Let's take 512 bits (64 bytes) for now. At 32 bits per pixel – the standard for depth buffers and still fairly common for render targets although rendering workloads are definitely shifting towards 64 bit/pixel formats – that's enough memory to fit data for 16 pixels in. Hey, that's funny – we're usually shading pixels in groups of 16 to 64! (NV is a bit closer to the smaller end, AMD favors the larger counts). In fact, the 8×8 tile size I've been quoting in the rasterizer / early Z parts comes from AMD; I wouldn't be surprised if NV did coarse traversal (and hierarchical Z, which they dub "Z-cull") on 4×4 tiles, though a quick web search turned up nothing to either confirm this or rule it out. Either way, the plot thickens. Could it be that we're trying to traverse pixels in an order that gives good DRAM page coherency? You bet we are. Note that this has implications for internal render target layout too: we want to make sure pixels are stored such that a single DRAM page actually has a useful shape; for shading purposes, a 4×4 or 8×2 pixel DRAM page is a lot more useful than a 16×1 pixel one (remember – quads). Which is why render targets usually don't have a fully linear layout in memory.

That gives us yet another reason to shade pixels in groups, and also yet another reason to do a two-level traversal. But can we milk this some more? You bet we can: we still have the memory latency to cover. Usual disclaimer: This is one of the places where I don't have detailed information on what GPUs actually do, so what I'm describing here is a guess, not a fact. Anyway, as soon as we've rasterized a tile, we know whether it generates any pixels or not. At that point, we can select a ROP to handle our quads for that tile, and queue a command to fetch the associated frame buffer data into a buffer. By the point we get shaded quads back from the shader units, that data should be there, and we can start blending without delay (of course, if blending is off or identity, we can skip this load altogether). Similarly for Z data – if we run early Z before the pixel shader, we might need to allocate a ROP and fetch depth/stencil data earlier, maybe as soon as a tile has passes the coarse Z test. If we run late Z, we can just prefetch the depth buffer data at the same time we grab the framebuffer pixels (unless Z is off completely, that is).

All of this is early enough to avoid latency stalls for all but the fastest pixel shaders (which are usually memory bandwidth-bound anyway). There's also the issue of pixel shaders that output to multiple render targets, but that depends on how exactly that feature is implemented. You could run the shader multiple times (not efficient but easiest if you have fixed-size output buffers), or you could run all the render targets through the same ROP (but up to 8 rendertargets with up to 128 bits/pixels – that's a lot of buffer space we're talking), or you could allocate one ROP per output render target.

An of course, if we have these buffers in the ROPs anyway, we might as well treat them as a small cache (i.e. keep them around for a while). This would help if you're drawing lots of small triangles – as long as they're spatially localized, anyway. Again, I'm not sure if GPUs actually do this, but it seems like a reasonable thing to do (you'd probably want to flush these buffers something like once per batch or so though, to avoid the synchronization/coherency issues that full write-back caches bring).

Okay, that explains the memory side of things, and the computational part we've already covered. Next up: Compression!

Depth buffer and color buffer compression

I already explained the basic workings of this in **part 7 (https://fgiesen.wordpress.com/2011/07/08/a-trip-through-the-graphics-pipeline-2011-part-7/)** while talking about Z; in fact, I don't have much to add about depth buffer compression here. But all the bandwidth issues I mentioned there exist for color values too; it's not so bad for regular rendering (unless the Pixel Shaders output pixels fast enough to hit memory bandwidth limits), but it is a serious issue for MSAA, where we suddenly store somewhere between 2 and 8 samples per pixel. Like Z, we want some lossless compression scheme to save bandwidth in common cases. Unlike Z, plane equations per tile are not a good fit to textured pixel data.

However, that's no problem, because actually, MSAA pixel data is even easier to optimize for: Remember that pixel shaders only run once per pixel, not per sample – unless you're using sample-frequency shading anyway, but that's a D3D11 feature and not commonly used (yet?). Hence, for all pixels that are fully covered by a single primitive, the 2-8 samples stored will usually be the same. And that's the idea behind the common color buffer compression schemes: Write a flag bit (either per pixel, or per quad, or on an even larger granularity) that denotes whether for all the pixels in a compression block, all the per-sample colors are in fact the same. And if that's the case, we only need to store the color once per pixel after all. This is fairly simple to detect during write-back, and again (much like depth compression), it requires some tag bits that we can store in a small on-chip SRAM. If there's an edge crossing the pixels, we need the full bandwidth, but if the triangles aren't too small (and they're basically never *all* small), we can save a good deal of bandwidth on at least part of the frame. And again, we can use the same machinery to accelerate clears.

On the subject of clears and compression, there's another thing to mention: Some GPUs have "hierarchical Z"-like mechanisms that store, for a large block of pixels (a rasterizer tile, maybe even larger) that the block was recently cleared. Then you only need to store one color value for the whole tile (or larger block) in memory. This gives you very fast color clears for some buffers (again, you need some tag bits for this!). However, as soon as any pixel with non-clear color is written to the tile (or larger block), the "this was just cleared" flag needs to be… well, cleared. But we do save a lot of memory bandwidth on the clear itself and the first time a tile is read from memory.

And that's it for our first rendering data path: just Vertex and Pixel Shaders (the most common path). In the next part, I'll talk about Geometry Shaders and how that pipeline looks. But before I conclude this post, I have a small bonus topic that fits into this section.

Aside: Why no fully programmable blend?

Everyone who writes rendering code wonders about this at some point – the regular blend pipeline a serious pain to work with sometimes. So why can't we get fully programmable blend? We have fully programmable shading, after all! Well, we now have the necessary framework to look into this properly. There's two main proposals for this that I've seen – let's look at the both in turn:

1. Blend in Pixel Shader – i.e. Pixel Shader reads framebuffer, computes blend equation, writes new output value.
2. Programmable Blend Unit – "Blend Shaders", with subset of full shader instruction set if necessary. Happen in separate stage after PS.

1. Blend in Pixel Shader

This seems like a no-brainer: after all, we have loads and texture samples in shaders already, right? So why not just allow a read to the current render target? Turns out that unconstrained reads are a *really* bad idea, because it means that every pixel being shaded could (potentially) influence every other pixel being shaded. So what if I reference a pixel in the quad over to the left? Well, a shader for that quad could be running this instant. Or I could be sampling half of my current quad and half of another quads that's currently active – what do I do now? What exactly would be the correct results in that regard, never mind that we'd probably have to shade all quads sequentially to reliably get them? No, that's a can of worms. Unconstrained reads from the frame buffer in Pixel Shaders are out. But what if we get a special render target read instruction that samples one of the active render targets at the current location? Now, that's a lot better – now we only need to worry about writes to the location of the current quad, which is a way more tractable problem.

However, it still introduces ordering constraints; we have to check all quads generated by the rasterizer vs. the quads currently being pixel-shaded. If a quad just generated by the rasterizer wants to write to a sample that'll be written by one of the Pixel Shaders that are currently in flight, we need to wait until that PS is completed before we can dispatch the new quad. This doesn't sound too bad, but how do we track this? We could just have a "this sample is currently being shaded" bit flag… so how many of these bits do we need? At 1920×1080 with 8x MSAA, about 2MB worth of them (that's bytes not bits) – and that memory is global, shared and determines the rate at which we can issue new quads (since we need to mark a quad as busy before we can issue it). Worse, with the hierarchical Z etc. tag bits, they were just a hint; if we ran out of them, we could still render, albeit more slowly. But this memory is *not* optional. We can't guarantee correctness unless we're really tracking every sample! What if we just tracked the "busy" state per pixel (or even quad), and any write to a pixel would block all other such writes? That would work, but it would massively harm our MSAA performance: If we track per sample, we can shade adjacent, non-overlapping triangles in parallel, no problem. But if we track per pixel (or at lower granularity), we effectively serialize all the edge quads. And what happens to our fill rate for e.g. particle systems with lots of overdraw? With the pipeline I described, these render (more or less) as fast as the ROPs can merge the incoming pixels into the store buffers. But if we need to avoid conflicts, we really end up shading the individual overlapping particles in order. This isn't good news for our shader units that are designed to trade latency for throughput, not at all.

Okay, so this whole tracking thing is a problem. What if we just force shading to execute in order? That is, keep the whole thing pipelined and all shaders running in lockstep; now we don't need tracking because pixels will finish in the same order we put them into the pipeline! But the problem here is that we need to make sure the shaders in a batch actually always take the exact same time, which has unfortunate consequences: You always have to wait the worst-case delay time for every texture sample, need to always execute both sides of every branch (someone might at some point need the then/else branches, and we need everything to take the same time!), always runs all loops through for the same number of iterations, can't stop shading on discard… no, that doesn't sound like a winner either.

Okay, time to face the music: Pixel Shader blend in the architecture I've described comes with a bunch of seriously tricky problems. So what about the second approach?

2. "Blend Shaders"

I'll say it right now: This can be made to work, *but…*

Let's just say it has its own problems. For once, we now need another full ALU + instruction decoder/sequencer etc. in the ROPs. This is not a small change – not in design effort, nor in area, nor in power. Second, as I mentioned near the start of this post, our regular "just go wide" tactics don't work so well for blend, because this is a place where we might well get a bunch of quads hitting the same pixels in a row and need to process them in order, so we want low latency. That's a very different design point than our regular unified shader units – so we can't use them for this (it also means texture sampling/memory access in Blend Shaders is a big no, but I doubt that shocks anyone at this point). Third, pure serial execution is out at this point – too low throughput. So we need to pipeline it. But to pipeline it, we need to know how long the pipeline is! For a regular blend unit, it's a fixed length, so it's easy. A blend shader would probably be the same. In fact, due to the design constraints, you're unlikely to get a blend shader – more like a blend register combiner, really, completely with a (presumably relatively low) upper limit on the number of instructions, as determined by the length of the pipeline.

Point being, the serial execution here really constrains us to designs that are still relatively low-level; nowhere near the fully programmable shader units we've come to love. A nicer blend unit with some extra blend modes, you can definitely get; a more open register combiner-style design, possibly, though neither the API guys nor the hardware guys will like it much (the API because it's a fixed function block, the hardware guys because it's big and needs a big ALU+control logic where they'd rather not have it). Fully programmable, with branches, loops, etc. – not going to happen. At that point you might as well bite the bullet and do what it takes to get the "Blend in Pixel Shader" scenario to work properly.

…and that's it for this post! See you next time.

From → Coding, Graphics Pipeline

7 Comments

1. **Aras Pranckevičius permalink**
   First things first: awesome post series!

   Now, onto the programmable blending ;) Do you have any information/intuition how some mobile GPUs (PowerVR SGX, NVIDIA Tegra 2, ARM Mali) do programmable blending? They do have it, and not all of them are tile based rasterizers.

   Reply
   - **fgiesen permalink**

Sorry for taking so long to reply, your comment ended up in the spam folder for some reason. Anyway, no, I don't have detailed information, but here's some thoughts:

On all mobile chips, clocks are much lower; memory latency is a bit higher too, but overall, memory wait times are still lower in terms of cycles. That helps. They also have a far lower count of shader units, which means "issue stalls" due to queued quads depending on quads that are still being shaded are less expensive overall (the throughput cost due to a stall is proportional to the number of units that are left idle). Finally, because they have both lower clocks and a less extreme memory:arithmetic latency ratio, they need less quads in-flight per shader to sustain good utilization.

To explain a bit more: The scoreboarding-based scheme I described is one way to avoid "shading races" (and the "most natural" if you're thinking in software terms). Another is to keep track of all in-flight quads. Then every new quad is tested against all in-flight quads for coverage mask collisions (this can be implemented by keeping track of the in-flight quads in a small content-addressable memory, aka CAM). If you have a reasonably small number (say between 16 and 64) quads in-flight at any given time, this works fine. But with 4-16 quads per batch, 10+ batches/Warps/Wavefronts running on a shader unit at any given time, and dozens of shader units, you need an impracticably large CAM (and they're power-hungry as hell!). And of course for tile-based renderers the whole scoreboard size issue in the direct scoreboarding algorithm disappears too.

You can make programmable blending work without a huge extra cost if you have fixed latencies (register combiners or shaders without dynamic branching), or if you keep the contested resource has a reasonably small bounded size (e.g. tile-based), or if the number of agents that can conflict is kept small (low number of in-flight quads). But when you have variable latency and impractically large bounds on render target size and number of in-flight quads, you're in trouble. :)

Reply

2. **Kevin Rogovin permalink**
Wonderful series, I wanted to comment on the blend shader thing in context of mobile.

1) For tiled based renderers, i.e. SGX, Mali, Adreno since the rasterization takes place on (tiny) tile at a time on SRAM, the entire memory pain of a blend shader does not exist. Indeed, there is an ES extension (that Apple iOS now supports) that allows one to read the value of the "framebuffer".

2) Also on mobile, but NOT a tiled based renderer: NVIDIA Tegra (2,3 and 4) also allow one to read the framebuffer value from the fragment shader. One of the icky things is that using the NVIDIA offline compiler, one needs to pragma the blend-state so that it will append those instructions to the fragment shader. I know Tegra is not tiled based, so this design decision I think is odd.

Reply

- **fgiesen permalink**
Yeah, my general discussion applies to both non-tiled and tiled renderers, and I was actually working on a shader compiler for a tiled renderer (with programmable blend inside shaders) at the time I wrote this. :)

The general issue stays the same, though: blending/late Z write is a synchronization point; for any given pixel in the render target, the blend/Z etc. operations have to happen in the right order. There's numerous ways to solve this and all have different trade-offs.

At the very least, the "blend" stages of pixel shading for quads hitting the same location in the render target need to run in the right order, and one way to provide this guarantee (and nothing stronger) is the scoreboarding-like scheme I describe. The problems I discuss wrt render target sizes don't exist in a tiled renderer; all this checking happens tile by tile, so the size is fixed and everything just works. You can also be even more strict and just require that *all* quads blend in the order they were rasterized; this avoids the need for bookkeeping but means that shaders now have a "blend barrier" right before blending starts. This is less bookkeeping but means pixel shader warps/wavefronts are "live" (and potentially stalled) for longer, reducing the available resources for other warps/wavefronts that could actually do useful work instead of just sitting around waiting for their turn at blending. How expensive this is depends on how much it reduces your utilization, which depends on lots of other things including the expected complexity of the shaders you're running.

For a non-tiled renderer like the Tegra, anything that requires explicit per-quad bookkeeping is icky (for the reasons described in the article), which means they're probably using a less precise scheme like the "in-order blend" stuff I described above. Now this doesn't mean that *all* blend operations inside the pipeline have to be synchronized against each other; for example, if you have 4 "shader cores" (the high-level ones, I'm not talking about "CUDA cores" or whatever NVidia's current nomenclature of the day is here), you can just assign a quarter of the render target to each core, usually in some kind of checkerboard pattern. Each shader core "owns" that part of the screen. With that kind of scheme, each shader core only needs to synchronize blending operations against other blending operations it's done by itself, there's no global "locks".

That's one way to make this kind of approach scale; it still suffers once you have complex shaders with several branches and very variable run times though, because one quad that takes long to shade can hold up blending for everything that happens after it.

The ROP design you see in high-end GPUs essentially does the same thing; each ROP owns part of the render target, so they don't conflict and don't need to talk to each other to do the right thing. Blending is still relatively serial work within a ROP. But you can have lots of them, and more importantly, a ROP stalling because some quads aren't done shading yet won't necessarily block the shader units, not until the ROP's input queue fills up anyway. So instead of stalling the shader units (which could be doing all kinds of other work in the mean time), you stall a dedicated unit whose only task is to blend, which has very little state per pixel (much less than the original shader would), and which is designed to be fast enough to "catch up" after most stalls without causing any hitches upstream (in the shader cores, which we want to keep busy).

Reply

Trackbacks & Pingbacks

1. A trip through the Graphics Pipeline 2011: Index « The ryg blog
2. A trip through the Graphics Pipeline 2011, part 13 « The ryg blog
3. Photoshop Blend Modes in Unity – The Code Corsair